INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tel (1) 39 63 55 11

Rapports de Recherche

N° 980

Programme 1

TIN : UN OUTIL GENERAL D'ETUDE ET DE COMPARAISON DES ANALYSEURS SYNTAXIQUES

Sylvie BILLOT

Février 1989

RR-0980

TIN : un outil général détude et de comparaison des analyseurs syntaxiques

Sylvie Billot

Résumé

Le système TIN décrit ici est un générateur d'analyseurs syntaxiques pour toute grammaire algébrique sans restriction. Il est basé sur la méthode décrite par B.Lang dans [10]. Cette méthode généralise celle d'Earley en séparant nettement les problèmes liés à la reconnaissance d'une structure syntaxique particulière d'après les règles de la grammaire traitée et ceux posés par la possible reconnaissance de plusieurs structures sur une même chaîne à analyser. La présentation faite ici de ce générateur permet de définir les bases de son implantation, et les différentes structures de données nécessaires (Transducteur à pile, item et grammaire résultat). La phase de construction du transducteur par lecture des règles de grammaire peut être réalisée grâce à n'importe quelle technique d'analyse syntaxique. TIN permet donc de comparer sur les mêmes bases formelles des techniques différentes. Cette comparaison est ici faite entre une méthode de simple précédence, une méthode Lalr(1) et une méthode descendante. Un manuel d'utilisation du système présenté est enfin donné en annexe.

TIN : a general tool for studying and comparing parsers

Sylvie Billot

Abstract

The Tin system described here is a parser generator for every context-free grammar, without any restriction. It is based on the method described by B.Lang in [10]. This method generalizes Earley's one by clearly separating problems setted by recognition of a particular syntaxic structure from the grammar and those setted by the possibility to get by non determinism several structures on a same input string. Our présentation of this method consists to define its implementation bases, and the different manipulated data structures (Push-Down Transducer, item and resulting grammar). The step of building the Transducer by reading the grammar may be done by any parsing technique. Comparisons of different techniques on the same formal bases may thus be done with TIN. Here, we show it by comparing a weak precedence, Lalr(1) and a recursive descent methods. A Reference Manuel is then given in appendix.

PAPIER RÉCUPÉRÉ ET RECYCLÉ

1 Introduction

L'activité de production de logiciels conduit au développement d'outils d'assistance de plus en plus perfectionnés. La mise au point de ces outils nécessite à la fois des études théoriques et expérimentales, ces dernières permettant de déterminer les problèmes posés par la pratique et de réaliser des prototypes basés sur une analyse théorique de ces problèmes. Le système TIN se situe dans ce contexte en étant un outil d'expérimentation de techniques d'analyse syntaxique non-déterministes. Ces techniques peuvent être généralisées à des cas plus complexes de phénomènes non déterministes, aussi TIN est un outil général d'étude de techniques de gestion de ces phénomènes.

Les langages utilisés pour modéliser l'aspect syntaxique d'une communication sont de la classe des langages dits *algébriques* ou *non contextuels*. Un résultat de base est qu'une phrase d'un tel langage peut être reconnue par un automate à pile. Cette machine est appelée **reconnaisseur** si le résultat d'un de ses calculs sur une phrase particulière permet seulement de déterminer si la phrase appartient au langage de l'automate ou non; elle est appelée **analyseur** si le résultat du calcul, lorsque la phrase est valide, est la représentation de sa syntaxe sous forme d'une structure d'arbre syntaxique.

De nombreux travaux ont été effectués dans ce domaine et il existe en particulier pour les langages de programmation des générateurs d'analyseurs syntaxiques très puissants comme Syntax([6]) ou Yacc ([16]). Ils génèrent cependant des analyseurs qui sont déterministes et se limitent au traitement de certaines classes de langage.

Les algorithmes plus généraux qui existent d'autre part ne sont pas utilisables pratiquement parce que le fait d'avoir à gérer du non-déterminisme les rend inefficaces. Parmi ces méthodes générales, la plus connue est celle d'Earley ([7]). Cette méthode connaît actuellement de nombreux développements ([14], [15]) car elle peut être améliorée par un calcul à l'avance de certaines structures de traitement.

L'algorithme sur lequel est basé TIN généralise cette idée de calcul préliminaire, une décomposition des traitements étant faite entre

- la construction des éléments de l'analyse
- et leur exploitation pour la gestion du non-déterminisme.

Cet algorithme, décrit de façon théorique par B.Lang ([10]), est validé ici par son application pratique. Elle a conduit à l'écriture d'un générateur d'analyseurs syntaxiques qui traite toutes les grammaires non contextuelles, y compris les grammaires cycliques.

Le but de ce papier est de montrer les bases du système TIN implanté. Après avoir présenté les principes généraux sous-jacents de ce générateur, nous étudierons ici chaque composant du calcul : le transducteur à pile, l'algorithme utilisé pour l'interpréter et la forêt résultat de cette interprétation sur une chaîne correcte. L'utilisation pratique de ce système sera ensuite illustrée par la comparaison qu'il permet d'effectuer entre plusieurs méthodes d'analyse syntaxique.

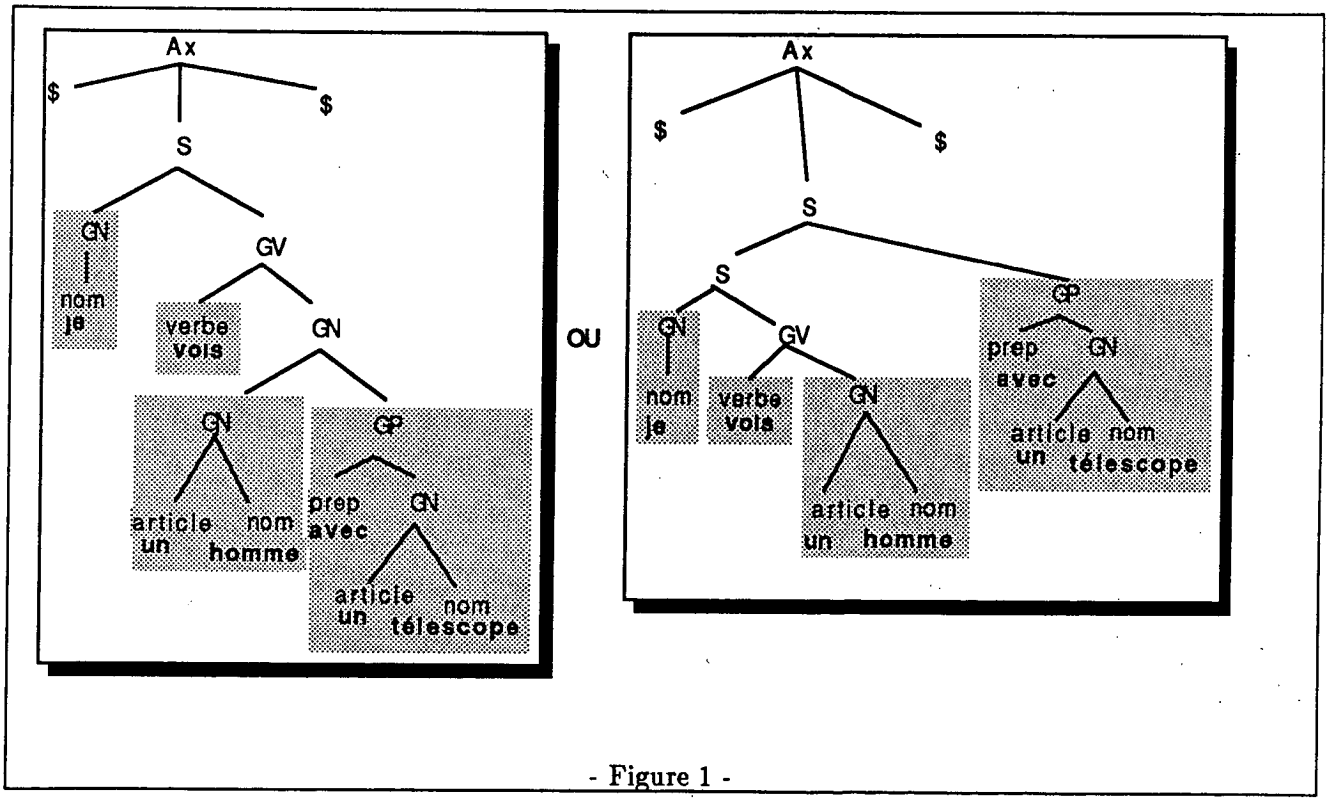
2 Généralités

Exemple :

Considérons la grammaire <N,T,P,Ax> dont l'ensemble de non-terminaux est N = {Ax,S, GN, GV}, celui de terminaux T = {article, nom, verbe}; l'axiome est Ax et l'ensemble de règles de production P :

(0) Ax ::= $ S $
(1) S ::= GN GV
(2) S ::= S GP
(3) GN ::= nom
(4) GN ::= article nom
(5) GN ::= GN GP
(6) GP ::= prep GN
(7) GV ::= verbe GN

La structure des phrases du langage décrit par cette grammaire suit les règles de P. Cette structure est représentée par un arbre syntaxique. Par exemple, la phrase *Je vois un homme avec un télescope* a deux arbres syntaxiques possibles (figure 1) Dans le premier, le groupe prépositionnel *avec un*



- Figure 1 -

télescope qualifie le groupe nominal *un homme.* Dans le deuxième arbre, ce groupe prépositionnel qualifie la phrase *je vois un homme.*

La grammaire de cet exemple est typiquement une grammaire ambiguë. La plupart des analyseurs existants, utilisés par les compilateurs de niveau industriel ne savent pas traiter de telles grammaires. Ils ne donnent pour résultat, s'ils en donnent un, qu'un seul arbre syntaxique, choisi de facon arbitraire. Pour traiter correctement le langage que l'on veut définir, il faut lever les am-

biguïtés en modifiant sa grammaire. Cet exercice est souvent difficile à faire, parfois impossible, comme dans l'exemple ci-dessus, sans perdre une partie de la richesse du langage.
Il existe des méthodes générales qui peuvent traiter les grammaires ambiguës et fournissent, pour une phrase donnée, son ensemble d'arbres syntaxiques, appelé **forêt.**

On distingue 2 grandes familles d'algorithmes généraux :

- Dans la première, les différents arbres syntaxiques sont construits en profondeur d'abord. Lorsqu'il y a non déterminisme, le choix est mémorisé afin de suivre chaque possibilité l'une après l'autre, par retour-arrière. Cette méthode est celle de l'algorithme de Cocke,Kasami et Younger ([5], [9], [18]) en particulier.

- Dans la deuxième, dont le représentant de base est l'algorithme d'Earley, tous les arbres syntaxiques possibles sont construits en parallèle. A chaque mot de la chaîne d'entrée sont associées toutes les structures valides jusqu'à ce mot, le choix de ces structures d'après les règles de grammaire étant limité par la lecture de la chaîne. Cette méthode de résolution, par application du principe d'optimalité, est appelée *programmation dynamique.* Dans la séquence optimale de décisions qu'est la construction d'un arbre syntaxique d'après les règles de grammaire, quelle que soit une décision prise (appliquer telle ou telle règle après lecture d'une partie de chaîne), les décisions subséquentes seront optimales d'après les résultats de la première décision. Ce qui rend ce type de démarche efficace est qu'on peut alors se contenter de ne mémoriser que les sous-politiques optimales (sachant que telle structure a été reconnue jusquà tel mot lu, on ne s'intéresse plus qu'au sous-problème de reconnaître la suite de la chaîne a partir de cette structure).

L'algorithme utilisé dans le générateur d'analyseurs syntaxiques présenté ici appartient à cette deuxième famille. Il est illustré par la figure 2 et a pour principale originalité de calculer à l'avance toutes les politiques de construction d'arbres possibles, afin de séparer ce calcul du choix non-déterministe à faire des arbres valides d'une chaîne particulière.

La recherche des éléments de l'analyse pour toutes les constructions d'arbres possibles est faite grâce à une méthode bien établie pour le cas déterministe. Dans le cas où la grammaire traitée n'est pas de la famille des grammaires acceptées par cette méthode déterministe, les conflits existants sont considérés comme étant des points de non-déterminisme du calcul. Ici, trois algorithmes particuliers ont été implantés :

- une méthode de simple précédence ([8]).

- Une méthode Lalr(1).

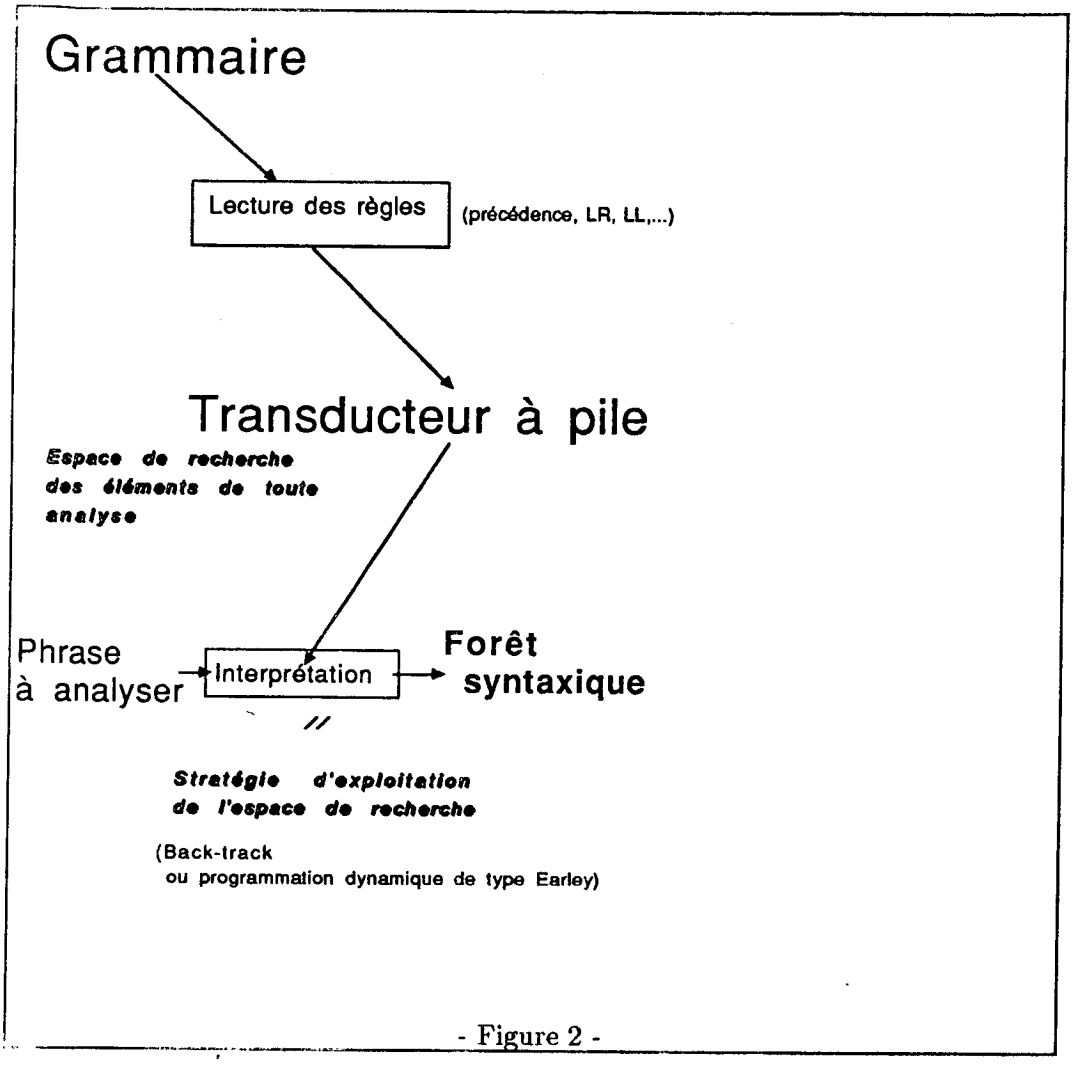
- Une méthode descendante récursive.

les deux dernières méthodes ayant été écrites d'après [1].

Le résultat de l'application d'une de ces méthodes à une grammaire particulière est un **Transducteur à pile**, c'est-à-dire un automate auquel est associée une fonction d'expression des structures reconnues. Ce transducteur peut être non déterministe si la grammaire n'est pas du type accepté par l'algorithme.

Un tel automate est alors interprété selon un algorithme unique dont la stratégie de gestion du non-déterminisme est celle d'Earley (construction en parallèle de tous les arbres syntaxiques possibles).

Le résultat de cette interprétation est une forêt dénotée sous forme de grammaire. Les non-terminaux de cette grammaire sont les calculs de construction des arbres, les terminaux étant



- Figure 2 -

les composants de ce calcul (mots lus dans la chaîne d'entrée et règles de la grammaire analysée reconnues). Cette représentation présente l'avantage de compacter le résultat sous forme d'arbres binaires, d'exprimer le partage de structures communes entre plusieurs arbres (reconnaissance du même non-terminal selon la même règle), ainsi que de dénoter les cycles du calcul. Ce dernier avantage a permis de développer un analyseur de phrases incomplètes décrit dans [11].

3 Le transducteur à pile

Le transducteur à pile représente tous les calculs possibles de reconnaissance de toutes les phrases du langage par une méthode d'analyse particulière. Son interpréteur est unique, quelle que soit sa méthode de construction.
Pour rendre cela possible, l'algorithme théorique ([10]) est basé sur la notion générale de transition ($\delta(p,B,a) \ni (r,A,u)$) qui fait le lien entre deux états p et r et entre deux symboles de pile A et B pour un élément a lu et un caractère u émis. Pour baser l'implantation sur le même concept, il faut fournir, pour chaque type d'automate, une fonction de lecture qui aura pour résultat une liste de transitions possibles pour un état de départ, un symbole de pile et un caractère à lire donnés.

Ce mécanisme, quoique général, semble coûteux et inutile. En effet, il fait intervenir deux niveaux d'interprétation : le premier pour calculer les transitions de l'automate, le deuxième pour effectuer l'analyse à partir de ces transitions. Pour éviter cela, l'automate peut être écrit directement en termes d'actions sur la pile, la chaîne d'entrée ou de sortie, effectuées à partir de chaque état. Si le langage de ces actions est unique et utilisé pour écrire tout automate, l'interprétation peut se faire à partir de chaque type d'action, qui correspond à un type de transition. Par exemple l'action de poser C sur la pile pour passer de l'état p à l'état r correspond à une transition $\delta(p,(),()) \ni (r,C,())$. La construction d'une structure supplémentaire, la transition, est ainsi évitée et il n'y a plus qu'un seul niveau d'interprétation. En contrepartie, ce langage doit décrire des actions assez élémentaires pour qu'elles gardent leur généralité et que tout automate puisse être interprété.

Par exemple une action telle que "lire une unité syntaxique, la comparer à celle qui est attendue (la variable) et, si elle est égale, la poser sur la pile " est décomposée en :

- un test "checkwindow" pour la comparaison de la variable et de la prochaine unité syntaxique à lire.
- une lecture "scan", c'est-à-dire appeler l'analyseur lexical pour passer à l'unité syntaxique suivante.
- une action "push" sur la pile.

Ainsi, dans une analyse topdown, qui pose sur la pile avant de lire, les mêmes actions pourront être utilisées, dans un autre ordre :

- une action "push" sur la pile.
- un test "checkwindow" pour la comparaison de la variable et de l'unité syntaxique à lire.
- une lecture "scan".

Chaque action de la chaîne écrite pour représenter l'automate est alors la structure unique à interpréter. Seule l'étape d'écriture de l'automate dans le langage des actions est nécessaire.

3.1 Le langage de description de l'automate

Un automate est un programme écrit dans le langage unique des actions élémentaires possibles. Chaque méthode de construction d'un automate écrit un programme particulier.

Une instruction d'un tel programme est composée d'informations qui sont :

- le nom de l'action.
- la variable sur laquelle s'exerce l'action.
- un booléen indiquant le non-déterminisme.
- le numéro de l'état où aller si le suivant ne convient pas (instructions de tests), ou à exécuter en parallèle si l'instruction est non-déterministe.

Les différentes actions sont regroupées dans le tableau ci-après :

NOM	ACTION	VARIABLE
scan	Emettre variable comme output et appeler l'analyseur lexical	Un terminal de la grammaire de départ.
push	Poser la variable sur la pile d'analyse.	Objet que l'on veut poser sur la pile (un symbole de la grammaire initiale, un état Lalr...).
checkstack	tester le haut de la pile.	une liste d'objets, dans laquelle on veut que se trouve le sommet de la pile.
checkwindow	tester si le résultat d'analyse lexicale est dans variable (contexte de longueur 1)	La liste des symboles possibles comme résultat d'analyse lexicale à ce moment de l'analyse.
pop	enlever le haut de la pile d'analyse.	*il n'y en a pas.*
output	émettre un terminal dans la grammaire résultat.	ce que l'on veut exprimer comme action lorsqu'on a reconnu un non-terminal de la grammaire de départ. Ici, c'est le numéro de la règle qui a permis la reconnaissance.
halt	Arrêt de l'analyse avec succès.	*il n'y en a pas.*
error	Arrêt de l'analyse en échec.	*il n'y en a pas.*
go	branchement.	numéro de l'état où se brancher.

EXEMPLE :

La reconnaissance des règles (3) et (4) de la grammaire de l'exemple de la section 2 peut être représentée par l'automate construit avec une méthode de précédence[1] :

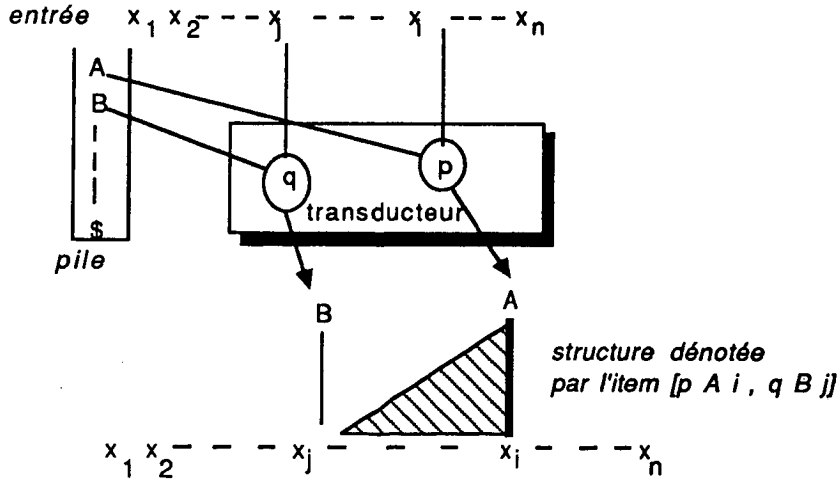
```
etat    nom           variable                  non-det    err
0       error
1       checkwindow   (article)                            34
2       scan          article
3       push          article
4       checkwindow   (nom)                                0
5       scan          nom
6       push          nom
7       checkwindow   (verbe,prep,findefichier)            0
```

[1]Le ? indique que l'instruction est non-déterministe.

```
8       checkstack    (nom)                    0
9       pop
10      output        3              ?         30
11      push          GN
.....................
30      checkstack    (article)                0
31      pop
32      output        4
33      go            11
34      checkwindow   (nom)                    0
35      scan          nom
36      push          nom
37      go            7
```

4 L'interprétation du transducteur

L'interprétation du programme écrit dans le langage présenté ci-dessus pour exprimer un transducteur particulier est faite selon un algorithme unique ([10]). Cette interprétation est basée sur la notion d'item qui est la structure de calcul. Un item [(pAi),(qBj)] représente le lien entre deux configurations du transducteur et mémorise le calcul effectué entre ces deux configurations pour construire un sous-arbre de père A sur la chaîne $x_j...x_i$ (figure 3) Après la lecture des mots de la



- Figure 3 -

chaîne d'entrée jusquà x_j, l'automate peut être dans un état q avec une pile dont le sommet est B. Le calcul entre x_j et x_i permet de poser A sur la pile au dessus de B et d'être dans l'état p. Le calcul de A est fait selon une structure particulière de sous-arbre construite d'après les règles de la grammaire de départ sur la sous-chaîne $x_j...x_i$. Pour le calcul de reconnaissance de la chaîne d'entrée, on peut alors se permettre de ne mémoriser que A et le fait que ce symbole se reconnait entre p et q, entre x_j et x_i. Ceci, avec le calcul en parallèle de toutes les possibilités engendrées par non-déterminisme, est une caractéristique de la méthode d'Earley et, plus généralement, permet

de faire le lien avec les techniques de programmation dynamique.

L'interprétation du transducteur consiste à générer de nouveaux items à partir de ceux déjà construits (l'item initial est constitué d'un état de départ du transducteur, de la pile vide et de l'indice de lecture 0). Les items sont générés en parallèle, toutes les structures valides jusquà un indice i de la chaîne d'entrée étant calculées avant de traiter l'indice i+1.

Pour exprimer le résultat de l'analyse, il faut, outre calculer, par génération d'items, les sous-arbres pouvant se construire sur la phrase à analyser, garder trace de ces structures reconnues. Ceci est fait en mémorisant leur construction, chaque item étant associé à la manière dont il a été généré. Cette association prend la forme de la définition d'une grammaire résultat, dont

- les non-terminaux sont les items calculés pour reconnaître la chaîne d'entrée,
- les terminaux sont les éléments construction des arbres syntaxiques (symboles lus et numéros de règles appliquées),
- l'axiome est l'item final de reconnaissance de toute la phrase
- et les règles de productions sont les règles de calcul de chaque item.

4.1 Implantation des items

Un item [(pAi),(qBj)]est composé de deux éléments de même structure. Cette structure, appelée **mode**, est un triplet composé de :

- un numéro d'état de l'automate, c'est-à-dire d'instruction dans le programme de description de cet automate.
- une valeur de haut de pile.
- un indice de position dans la chaîne d'entrée.

Un mode (pAi) définit une configuration de l'automate sur laquelle s'appliquent les actions indiquées par p, sur une pile dont le symbole du haut a la valeur A alors que l'unité syntaxique d'indice (i+1) peut être lue.
Un item [(pAi),(qBj)]permet de mémoriser le calcul qui s'est effectué pour passer de la configuration (qBj) à (pAi). Ce calcul a fait intervenir un certain nombre de transitions pour reconnaître tous les sous-arbres syntaxiques qui peuvent se construire sur la sous-chaîne $x_j \cdots x_i$.

Un item étant un lien entre deux modes, pour ne pas dupliquer les modes à chaque liaison, l'idée principale de la structure choisie est de baser l'implantation d'un item sur la notion de mode. Un mode peut être premier mode de différents items. L'ensemble de ces items est appelé une **classe d'items de même premier mode**. Un item est alors défini par l'appartenance de la classe de son second mode à la classe de son premier mode.

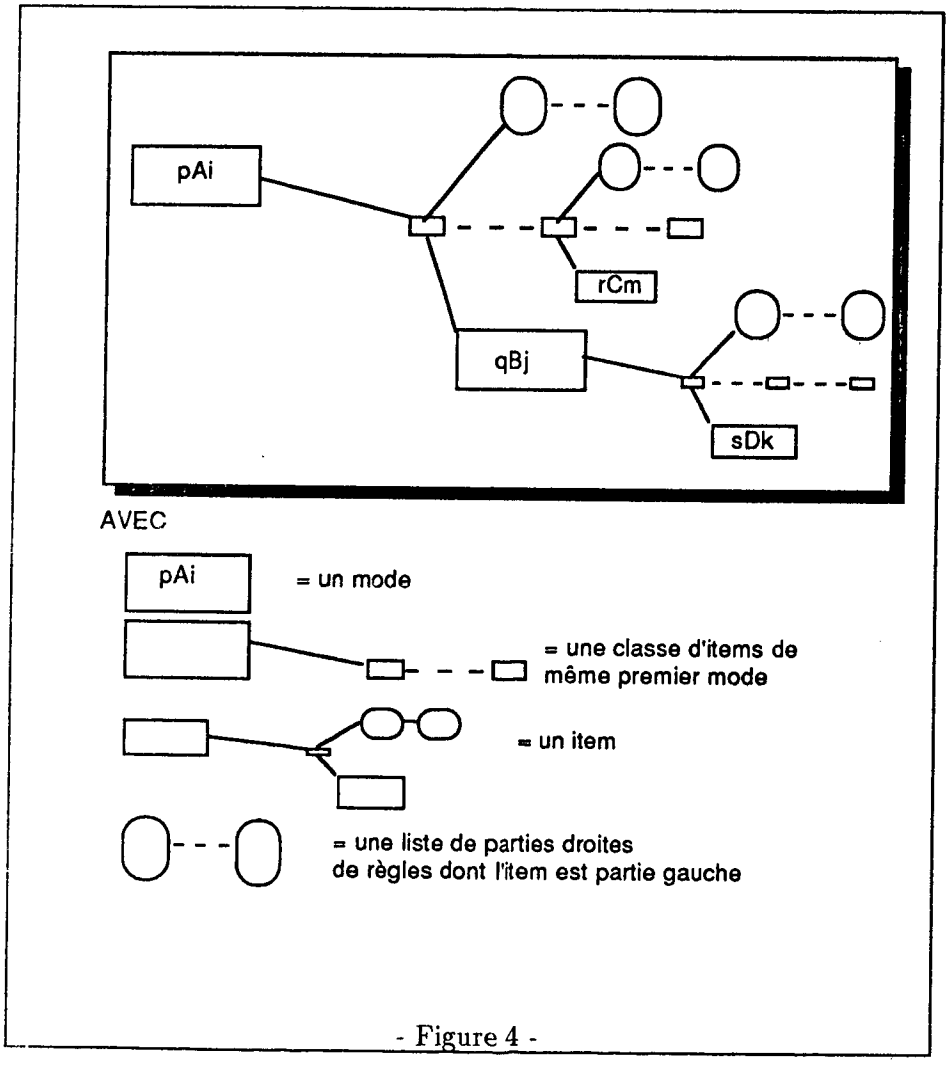
D'autre part, un item est à la fois une étape du calcul et un non-terminal de la grammaire résultat. Dans ce dernier rôle, il peut être représenté par la liste des parties droites des règles dont il est partie gauche. La grammaire résultat est écrite durant le calcul et seules les règles qui ont permis de construire les items terminaux doivent être prises en compte : en effet, seuls ces items représentent les calculs de reconnaissance de toute la phrase analysée. Afin d'éviter de garder des règles inutiles, la grammaire résultat est représentée grâce aux items : c'est à chaque item construit

qu'est associé l'ensemble de ses règles de construction. La grammaire résultat est alors l'ensemble de toutes les sous-grammaires ainsi associées à chaque item final.

Pour mémoriser l'ensemble G des règles de construction de chaque item, la classe de son second mode est associée à G dans la classe de son premier mode. Une classe d'items est donc composée des informations suivantes :

- le premier mode commun à tous les items.
- une liste de paires, chaque paire étant composée de
  - la classe d'items du second mode d'un item.
  - la liste des règles de construction de cet item.

La figure 4 représente la classe d'items de même premier mode (pAi) c'est-à-dire les items [(pAi),(qBj)],



- Figure 4 -

[(pAi),(rCm)], etc... ainsi que les items de même premier mode (qBj) : [(qBj),(sDk)], et ainsi de suite.

4.1.1 construction de nouveaux items

La forme des items générés et leur règle de construction dépendent de l'instruction correspondant à l'état courant du transducteur. Cette génération suit l'algorithme suivant [2] :

ANALYSE D'UNE CHAINE $X_1 X_2 \cdots X_i \cdots X_n$:

1. INITIALISATION :
   U_0 := [1 () 0),()] est l'item initial.
   Sa règle de construction est U_0 ::= ()

2. ITERATION :
   Pour i=0 à n-1, faire
   Pour chaque item U = [(pAi),(qBj)]de S_i associé à l'ensemble G de ses règles de construction),interpréter l'instruction numéro p du transducteur pour générer de nouveaux items et les ranger dans S_i ou S_{i+1} selon le cas.

Interprétation d'une instruction (nom,var,nondet,err) suivant son nom :

- *nom = scan*
  Ranger dans S_{i+1} le nouvel item
  - *[(p+1Ai+1),(qBj)]*
  - *règle de la construction = (G var)*
- *nom = push*
  Ranger dans S_i le nouvel item :
  - *[(p+1 var i),(pAi)]*
  - *règle de la construction = (nil)*
- *nom = checkstack*
  - *Si A ∈ var, interpréter l'instruction de numéro p+1.*
  - *sinon interpréter l'instruction de numéro err.*
- *nom = checkwindow*
  - *Si l'unité syntaxique lue (X_{i+1}) ∈ var, interpréter l'instruction de numéro p+1.*
  - *sinon interpréter l'instruction de numéro err.*
- *nom = pop*
  Pour chaque élément (classe (sDk),grammaire Y) de la liste d'items de la classe (qBj), ranger dans S_i le nouvel item
  - *[(p+1Bi),(sDk)]*
  - *règle de la construction = (Y G)*

  Si i = j, il se peut que certains items de la classe (qBi) ne soient pas encore construits. Pour pouvoir les traiter à leur création, il faut mémoriser dans une **listarevoir** *un lien entre l'état p+1, le mode (qbj) et la grammaire G.*
- *nom = output*
  Ranger dans S_i le nouvel item :

[2] S_i est l'ensemble des items associés à X_i

- *[(p+1Ai),(qBj)]*
- *règle de la construction = (G var).*

- *nom = go*
  Interpréter l'instruction de numéro var.

Les instructions de nom halt ou error ne donnent lieu à aucun calcul.

Dans tous les cas, si nondet est vrai (l'instruction est non-déterministe), il faut également interpréter l'instruction de numéro err (si ce n'est pas fait à cause du test pour les instructions checkwindow et checkstack)

3. TERMINAISON

Pour chaque item U de S_n :
Si U est final (de la forme $[f\ ()\ n), ()]$, avec f = instruction de nom "halt"), alors mettre les règles associées à U dans les règles de la grammaire résultat.
Sinon, interpréter l'instruction indiquée par U.

Le résultat de l'analyse est la grammaire des arbres syntaxiques de la chaîne analysée.

5 La forêt résultat

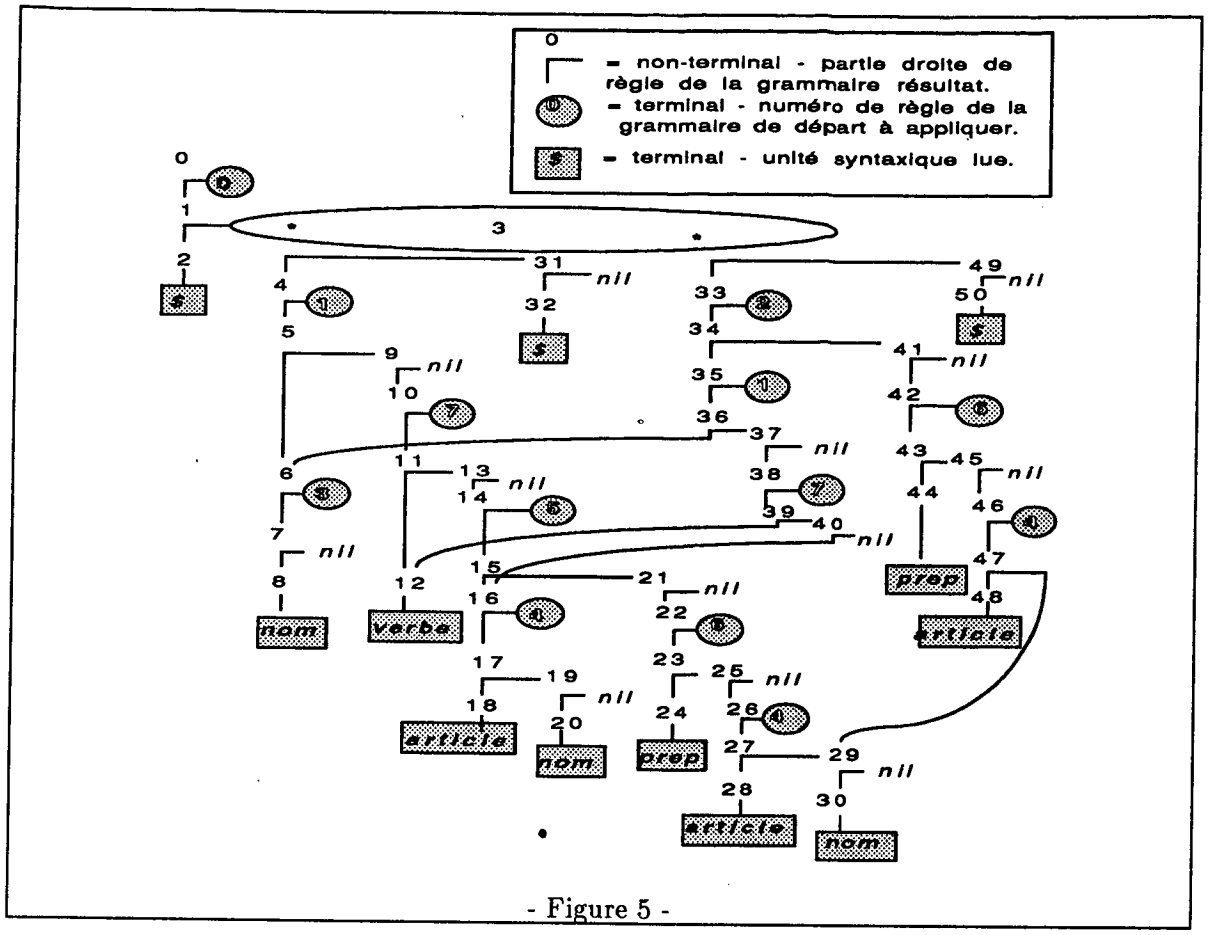
C'est une grammaire indépendante du contexte.
Les terminaux de cette grammaire sont les éléments de sortie du transducteur à pile. Dans le système ici implanté, ils sont de deux types:

- soit un numéro de règle de la grammaire en entrée, qui a été reconnue par une réduction.
- soit un élement lu de la chaîne analysée.

Les non-terminaux dans une partie droite de règle sont les items qui, d'après le calcul, sont à l'origine de la création de la partie gauche de la règle. Ces items sont ici considérés dans leur rôle de non-terminaux et sont des listes de parties droites de règles.

Chaque non-terminal est représenté par la liste des parties droites des règles dont il est partie gauche. Il y a **non-déterminisme** lorsqu'un non-terminal est partie gauche de plus d'une règle, c'est-à-dire lorsqu'un item peut se construire de plusieurs façons différentes.
Par exemple, la figure 5 représente la forêt d'analyse de la phrase "je vois un homme avec un télescope", le transducteur étant construit par une méthode Lalr(1) d'après les règles de la grammaire présentée dans la section 2. Les premières règles de cette grammaire résultat sont :

```
nt0 ::= nt1 0
nt1 ::= nt2 nt3
nt2 ::= $
nt3 ::= nt4 nt31                    nt3 ::= nt33 nt49
nt4 ::= nt5 1
etc ......
```



- Figure 5 -

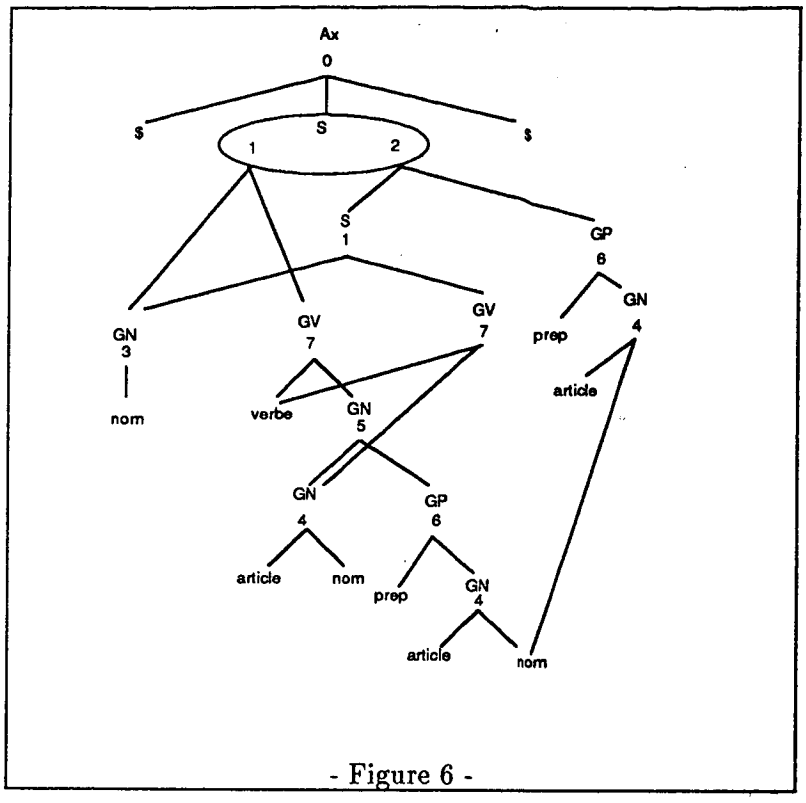
C'est à dire que l'item final, appelé ici nt0, a pour liste de parties droites de règles dont il est partie gauche une liste à un seul élément composé de la liste des règles dont l'item appelé nt1 est partie gauche et du terminal 0, numéro de la règle de la grammaire analysée à appliquer ((0) Ax ::= $ S $) après construction de nt1 ...
Les règles de cette grammaire résultat sont les règles de construction de tous les arbres syntaxiques de la chaîne reconnue.

Le non-déterminisme, qui conduit à la construction de deux arbres syntaxiques pour cette phrase, se traduit par l'existence de deux règles possibles de construction de l'item nt3. Les deux arbres syntaxiques de cette forêt sont représentés par la figure 6 .

6 Comparaison de méthodes déterministes

L'automate en entrée de l'analyseur peut être construit par n'importe quelle méthode déterministe d'analyse syntaxique. Les conflits détectés par cette méthode sont alors des points de non-déterminisme dans l'automate.
L'analyseur qui interprète cet automate, pour une chaîne donnée est ensuite le même, quel que soit l'automate et donc quelle que soit la méthode utilisée pour le construire. Ceci permet une comparaison de différentes techniques d'analyse.

Les méthodes qui ont été implantées pour construire l'automate en entrée sont *la simple*



- Figure 6 -

précédence, *Lalr(1)* et une méthode descendante (*top-down*).

Pour la grammaire de la section 2, les temps de construction de l'automate et le nombre d'instructions du code qui le représente sont respectivement :

précédence = 1,63 s - 81

Lalr = 1,01 s - 104

topdown = 0,12 s - 111

tests effectués sur les chaînes :

ch1 = "une femme chez elle écoute mozart"

ch2 = "je vois un homme avec un télescope"

ch3 = "je vois un homme avec un télescope dans le parc"

ch4 = "je vois un homme avec un télescope à la main dans le parc"

chaîne	temps d'analyse			items calculés			items utiles			forêt		
	p	l	td	p	l	td	p	l	td	p	l	td
ch1	0,15	0,08	0,38	88	47	184	47	47	77	31	31	27
ch2	0,20	0,14	0,53	94	77	225	73	73	111	51	51	40
ch3	0,43	0,33	0,91	194	155	367	133	139	190	99	103	72
ch4	0,75	0,58	1,37	343	272	542	215	236	289	169	189	116

avec :

p = précédence

l = Lalr

td = top-down

temps en secondes cpu

item calculés = nombre d'items construits
items utiles = nombre d'items intervenant dans la forêt résultat
forêt = nombre d'items nécessaires à exprimer cette forêt, après simplification de la grammaire.

tests effectués sur des chaînes de la forme nom verbe nom (prep nom)* , en faisant varier le nombre de "prep nom". ainsi, le non-déterminisme est augmenté de façon uniforme, en s'appliquant sur les mêmes règles de la grammaire de départ :

nb(prep n)	items calculés			items utiles			forêt		
	p	l	td	p	l	td	p	l	td
1	72	71	219	67	67	105	47	47	36
2	141	146	358	124	130	181	93	97	66
3	245	260	530	203	218	277	161	172	108
4	384	414	737	303	330	393	250	271	160
5	560	611	981	424	466	529	360	394	222

Les items calculés qui ne sont pas utiles représentent les essais d'analyse qui n'ont pas abouti. Lorsque la chaîne analysée n'est pas ambiguë (ch1), la technique Lalr calcule exactement le nombre d'items utiles, alors que les autres méthodes suivent des chemins inutiles.
Pour des chaînes ambiguës, les méthodes bottom-up (Lalr et précédence) calculent toujours moins d'items que la méthode top-down. Cependant le coefficient d'augmentation avec l'ambiguïté du nombre d'items calculés est moins important en top-down : cette méthode calcule de toutes façons beaucoup d'items dès le départ (chaîne déterministe ch1) et le nombre d'items supplémentaires nécessaires à gestion du non-déterminisme est ensuite proportionnellement moins important que lorsque l'automate est construit par une méthode bottom-up.

La méthode Lalr semble donc plus efficace au niveau des calculs, mais son résultat s'exprime avec plus de non-terminaux et prend donc plus de place en mémoire.
En particulier l'expression de la forêt (grammaire résultat dont on a enlevé toutes les règles inutiles, par exemple les règles vides ou celles de simple transmission de non-terminaux) nécessite moins de terminaux en top-down, et moins en précédence qu'en Lalr. Les calculs qui auront cette forêt pour base (sémantique) en seront affectés, en particulier si il y a beaucoup d'ambiguïté.

Pour les phrases ambiguës, la méthode de précédence partage naturellement mieux que le Lalr : elle calcule moins d'items et la forêt résultat s'exprime avec moins de non-terminaux. Ces deux méthodes suivent une stratégie bottom-up et se différencient par les tests effectués pour abandonner au plus tôt les calculs inutiles. Plus il y a dans la méthode utilisée de tests destinés à améliorer le déterminisme des choix à effectuer, moins le résultat est compact.

L'expérimentation de plusieurs schémas de compilation montre donc que la sophistication du constructeur d'automate peut avoir des effets négatifs sur l'efficacité de l'analyse non déterministe. La sophistication des analyseurs déterministes tend en effet à multiplier le nombre de cas particuliers, ce qui a une influence sur la taille du code du transducteur ainsi que sur le partage des calculs qui seront différenciés d'après leurs contextes d'analyse. Un bon moyen d'améliorer ce partage et donc souvent de réduire le temps de calcul, peut être de partager le plus possible les instructions du code de l'automate, même au prix d'une augmentation du non déterminisme. Cependant, cette augmentation du non déterminisme peut finalement se révéler plus coûteuse que le manque de partage des calculs. C'est le cas ici où la méthode Lalr améliore les performances du calcul par une meilleure détection au plus tôt des impasses.

Le choix d'un schéma d'analyse dépend donc de la grammaire traitée, du type de phrases à analyser et d'un bon équilibre à trouver entre l'efficacité des calculs et leur partage. TIN est un outil qui permet d'effectuer ce choix en se basant sur des résultats expérimentaux comparables.

7 conclusion

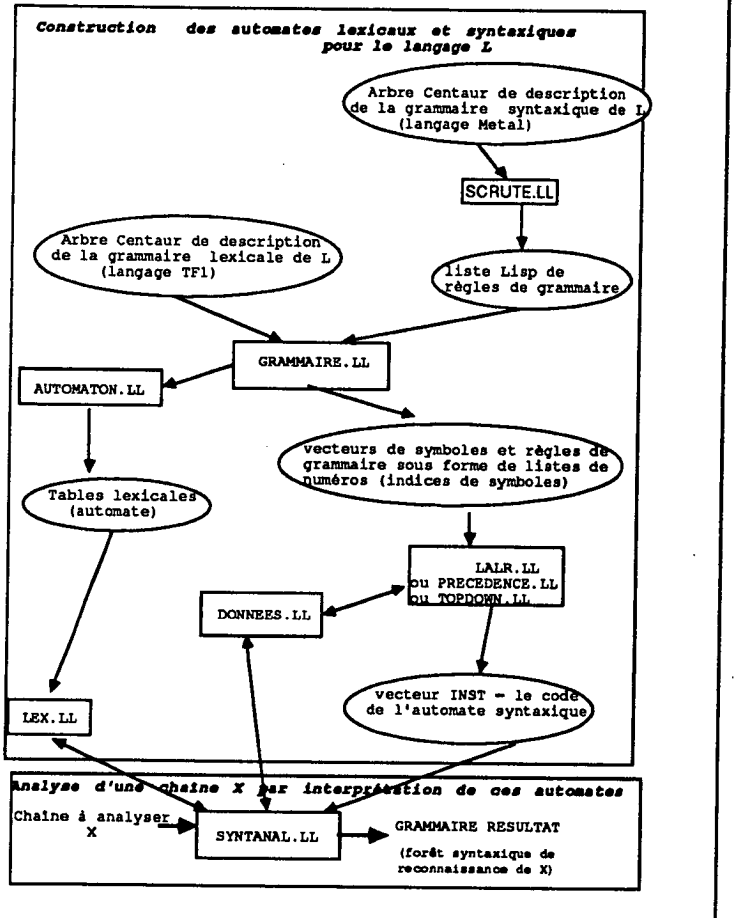
Le travail présenté ici a consisté à partir d'un modèle théorique et à le valider par une mise en oeuvre pratique et l'enrichir d'une expérimentation. La démarche expérimentale suivie vise à établir une formalisation des outils et des principes nécessaires au traitement du non-déterminisme en Analyse Syntaxique.
Le prototype réalisé permet de choisir selon les mêmes critères de base le type d'analyseur adapté au langage traité. En effet la base du calcul est ici non la grammaire mais un automate à pile dont la méthode de construction peut être choisie selon le résultat désiré qui peut être par exemple de disposer de forêts les plus compactes possibles, même au prix d'un calcul plus important. Ce choix n'existe pas dans la génération classique d'analyseurs, la méthode d'analyse étant déterminée une fois pour toutes par le générateur utilisé.

L'algorithme d'interprétation, en limitant les calculs, résumés dans la structure d'items, et en représentant les résultats sous forme d'une grammaire, formalisme qui permet en particulier de dénoter de manière finie les phénomènes éventuellement infinis, est un outil qui rend possible le traitement des phrases incomplètes ([11]).

Cet algorithme consiste à gérer le non-déterminisme sur une structure d'automate à pile. Les principes sous-jacents de modularité, d'indépendance des calculs qui concernent la construction des éléments de l'analyse et ceux qui concernent l'exploitation non-déterministe de ces éléments peuvent être généralisés à des structures plus complexes, comme celles d'Augmented Transition Network, ou encore celles utilisées dans la requête dans les bases de données ([12]) ou l'évaluation de programmes logiques ([13]).

A Manuel d'utilisation de TIN

Dans cette annexe est décrite l'utilisation de la version de TIN présentée. Une autre version a été écrite pour étudier l'amélioration du partage et est décrite dans [3]. Le schéma général des traitements est donné par la figure 7 . L'implantation qui a été faite s'appuie sur le système



- Figure 7 -

CENTAUR ([4]) pour la description de la grammaire traitée et la partie lexicale. La première chose à faire est de se mettre sous CENTAUR au niveau Lisp.

A.1 Fabriquer une grammaire d'entrée à partir d'un polish

Si **LANG.po** est le polish de description de la grammaire syntaxique du langage LANG en metal, il doit se trouver dans le directory courant.
Charger le fichier *scrute.ll.*
Exécuter la commande
(setq gsynt (transfgram (charger 'LANG ".")))
pour obtenir la grammaire sous forme d'une liste de règles, chaque règle étant une liste de car sa

partie gauche et cdr sa partie droite.
Pour afficher cette grammaire, charger le fichier *bnf.ll* et exécuter la commande (bnf gsynt).

A.2 Fabriquer l'arbre de description de la grammaire lexicale

Utiliser le langage TF1 ([2]) pour écrire le fichier **lang.tfl** qui décrit la grammaire lexicale du langage LANG.
Charger le fichier *tf1.ll.*
Exécuter la commande
(setq glex (parser 'lang tfl "."))
pour construire l'arbre metal de description de la grammaire lexicale.

A.3 travaux préliminaires

Charger le fichier *grammaire.ll.*
Exécuter la commande
(setq g (numgram glex gsynt)).
Cette commande appelle automaton (glex entries), entries étant la liste des mots clés de gsynt. Cette fonction permet de construire les tables lexicales.
La fonction numgram construit les vecteurs **vnonterm** et **vterm** qui contiennent respectivement les non terminaux et les terminaux de gsynt. Ceci permet d'attribuer un numéro à chaque symbole de la grammaire suivant la règle :
Un non terminal a pour numéro son indice dans **vnonterm.**
Un terminal a pour numéro la somme de la longueur du vecteur **vnonterm** et de son indice dans **vterm.**
Le résultat g est la grammaire gsynt à laquelle on a rajouté une règle de la forme ($ax $ axiome $) par axiome et décrite sous forme d'une liste de règles, chaque règle étant une liste de numéros de symboles.
Pour afficher cette grammaire en retrouvant les symboles, charger *donnees.ll* et *bnftrad.ll* et exécuter (bnftrad g).

A.4 Produire l'automate à pile en entrée

Charger le fichier *donnees.ll*, si ce n'est déjà fait. Suivant la méthode désirée, charger le fichier *precedence.ll lalr.ll* ou *topdown.ll.*
Dans les cas précédence ou Lalr, charger *bitmat.ll*, le module de traitement des matrices de bits.
Exécuter la commande **(precedence g), (lalr g) ou (topdown g)** pour obtenir le vecteur **INST** du code de l'automate.
La commande ({inst}:writcode inst) permet d'afficher ce code.

A.5 Analyse syntaxique

Charger les fichiers *lex.ll* (analyseur lexical) et *syntanal.ll* Positionner les booléens suivants, à () par défaut :
debugrules est à t si on veut voir apparaître la structure de calcul
[(pAi),(qBj)]de chaque non-terminal dans la grammaire résultat.
DEBUG est à t si on veut une trace de chaque item rangé.

Pour faire l'analyse syntaxique du fichier chaîne :
Exécuter la commande **(syntanal chaîne x)** x ayant pour valeur () pour une analyse purement non-déterministe, t pour un traitement du déterminisme local.

Si il n'y a pas d'erreur de syntaxe, le résultat est une grammaire mise sous forme de liste des parties droites des règles dont la partie gauche est un item final. Chaque partie droite est composée de terminaux (les output du code) et de non terminaux représentés par la liste des parties droites des règles dont ils sont partie gauche.
Cette grammaire peut être affichée, sous différentes formes de plus en plus simplifiées en exécutant la commande
({output}:imprimer grammaire).

A.6 Version simplifiée

Pour ne pas utiliser CENTAUR et travailler directement en LeLisp, on peut, pour des petites grammaires, écrire directement gsynt comme liste de règles (remplace l'étape 2). Cette liste est de la forme :
(nomgram $regle_1$ ... $regle_n$)
où nomgram est le nom que l'on donne à la grammaire (sert dans les procédures d'affichage), et chaque $regle_i$ = une liste (ntgauche partie droite) avec partie droite = une suite de symboles du vocabulaire, différenciés par le fait que l'on quote les symboles non terminaux.
Exemple: La grammaire
A ::= AA
A ::= a
s'écrira (gexemple ('A 'A 'A) ('A a))
Pour pouvoir utiliser les fonctions de définitions de données (record), il faut charger *cx.ll.*
Il faut ensuite redéfinir la fonction automaton : (de automaton (x y) ()) pour pouvoir commencer à l'étape 4 par : **(setq g (numgram () gsynt))** sans construire d'automate lexical.
Dans ce cas l'appel à syntanal se fera sur une chaîne résultat d'une analyse lexicale faite à la main. Par exemple, la chaîne "je vois un homme avec un télescope", pour la grammaire des prépositions sera entrée sous la forme (n v det n prep det n).
Cette chaîne sera lue en chargeant le fichier *simulex.ll* à la place de lex.ll.

References

[1] Aho, A.V.; Sethi, R.; and Ullman, J.D. 1986.
Compilers — Principles, Techniques and Tools.
Addison-Wesley.

[2] Battu, C. 1986.
Conception d'un générateur d'analyseur lexical sous l'environnement Mentor.
Mémoire d'ingénieur en Informatique. Conservatoire National des Arts et Métiers, Paris (France).

[3] Billot, S. 1988.
Analyseurs Syntaxiques et Non-Déterminisme. Thèse de Doctorat, Université d'Orléans, and INRIA (France).

[4] Borras, P.; Clément, D.; Despeyroux, T.; Incerpi, J.; Kahn, G.; Lang, B.; and Pascual V. Décembre 1987.
"CENTAUR: the system". Rapport de recherche INRIA numéro 777

[5] Cocke, J.; and Schwartz, J.I., 1970
Programming Languages and Their Compilers. Courant Institut of Mathematical Sciences, New-York University.

[6] Boullier, P. 1984.
Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques [3]. Thèse d'Etat, Université d'Orléans. France.

[7] Earley, J. 1970.
An Efficient Context-Free Parsing Algorithm.
Communications ACM 13(2): 94-102.

[8] Ichbiah, J.D.; and Morse, S.P. 1970.
A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars. *Communications ACM* 13(8): 501-508.

[9] Kasami, T 1965.
An efficient recognition and syntax analysis algorithm for context-free languages,
Sci-rep.AF CRL-65-758,Air Force Cambridge Research Laboratory, Bedford,Mass.

[10] Lang, B. 1974.
Deterministic Techniques for Efficient Non-deterministic Parsers. *Proc. of the* 2^{nd} *Colloquium on Automata, Languages and Programming,* J. Loeckx (ed.), Saarbrücken, Springer Lecture Notes in Computer Science 14: 255-269.
Also : Rapport de recherche IRIA numéro 72 (Mai 1974).

[11] Lang, B. 1988.
Parsing Incomplete Sentences.*Proc. of the* 12^{th} *Internat.Conf. on Computational Linguistics (COLING'88)* Vol.1 : 365-371, D.Vargha (ed.), Budapest (Hungary).

[3] SYNTAX est une marque déposée de l'INRIA

[12] Lang, B. 1988.
Datalog Automata. *Proc. of the 3rd Internat.Conf. on Data and Knowledge Bases*, C.Beeri, J.W. Schmidt, U. Dayal (eds.), Morgan Kaufmann Pub.,pp 389-404, Jerusalem (Israel).

[13] Lang, B. 1988.
Complete Evaluation of Horn Clauses, an Automata Theoritic Approach. INRIA Research Report (to appear).

[14] Chailloux, J.; Devin, M.; Dupont, F.; Hullot, J.M.; Serpette, B.; and Vuillemin, J. 1986
Le-Lisp, Version 15.2. Le manuel de référence. 3ème édition. Novembre 1986, INRIA, France.

[15] Tomita, M. 1987.
An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2): 31-46.

[16] Voisin, F. 1988.
Une version ascendante de l'algorithme d'Earley. PLILP88. Proc. of the International Workshop on Programming Languages Implementation and Logic Programming. May 16,18 1988. Orléans, France.

[17] *"YACC", UNIX* [4] *Programmer's Manuel, 4.3* Berkeley Software Distribution, April 1986.

[18] Younger, D.H. 1967.
Recognition of context-free languages in time n^3. *Inf. and Control* 10,2. 189-208

[4] UNIX est une marque déposée des BELL Laboratories